(2)

CMS Reverse Engineering

&

Encore/Model Integration

Contract # N00014-91-C-0240
Office of Naval Research
Arlington Virginia 22217-5000

Data Item A001

Bi-Monthly Progress Report

Reporting Period

September 1, 1991 - October 31, 1992

General Electric Company
Corporate Research and Development
P.O. Box 8
Schenectady, New York 12301

DISTRIBUTION

Mr. James G. Smith
Office of Naval Research
800 North Quincy Street
Arlington, VA 22217-5000
Attn. JGS, Code 1211
Ref: N00014-91-C-0240
(Scientific Officer)

DCMAO Hartford
130 Darling Street
East Hartford, CT 06108-3234
(Administrative Contracting Officer)

Director, Naval Research Laboratory
ATTN: Code 2627
Washington DC 20375

Defense Technical Information Center
Building 5, Cameron Station
Alexandria VA 22304-6145
(2-copies)

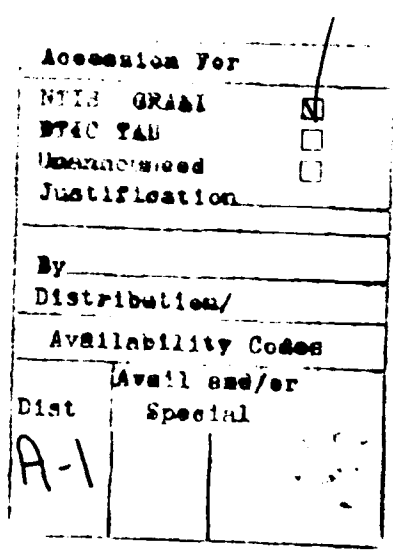
Mr. Ali Farsi
Code G042
Naval Surface Warefare Center
10901 New Hampshire Avenue
Silver Spring, MD 20903-5000

Mr. Lambert C. McCullough
Department of the Navy
Office of the Chief of Naval Research
Arlington, VA 22217-5000
(Contracting Officer)

Ms. Tamra Moore
Code U033
Naval Surface Warefare Center
10901 New Hampshire Avenue
Silver Spring, MD 20903-5000

Mr. Phillip Q. Hwang
Code U033
Naval Surface Warefare Center
10901 New Hampshire Avenue
Silver Spring, MD 20903-5000

Lt. Barry Stevens
Code 6113
Department of the Navy
Fleet Combat Direction Systems
Support Activity, Dam Neck
Virginia Beach, VA 23461-5300



Statement A per telecon
James Smith
ONR/Code 1267
Arlington, VA 22217-5000
NWW 12/5/91

TO: Distribution

FROM: Margaret Kelliher

DATE: October 31, 1991

SUBJECT Bi-Monthly Status Report - Contract # N00014-91-C-0240

---

1. Task 1: Language Processing and Analysis

In order to assure the quality and validity of the CMS-2 grammar being used to extract information for Reverse Engineering, an effort has been made to collect a number of "typical" source code examples. Currently, we are in possession of the following CMS-2 and CMS-2Y code. (CMS-2Y is a dialect which is compatible with "standard" CMS-2.)

A total of 94 files (52,000 lines of source code) has been received from NSWC which constitutes unclassified portions of a Mark 116 Mod 7 Mission Support Computer Program.

Three source modules have been received from Feet Combat Direction Systems, Dam Neck, as follows:

Module - 1 consisting of two files containing 5,600 lines of CMS-2Y source code. This module is a portion of the common system module in an FFG 7 class operational program which maintains an intercomputer interface.

Module - 2 consisting of six files containing 17,500 lines of CMS-2 source code. This module is described as being a WSN-5 inertial navigation simulation program.

Module - 3 consisting of twenty three files containing 3,000 lines of CMS-2Y source code. This module is described as being a portion of the training function of the CG/DDG ADCS Block 0 program.

We have successfully tested our parser against (minimally edited) examples from each of these systems. We are currently working on the design for COMPOOL handling. We do not currently support the following constructs:

CSWITCH - conditional compilation directives

MEANS and EXCHANGE - macro substitution directives

COMPOOL - Task 1 is currently addressing Compool processing

No effort will be made to deal with CSWITCH, MEANS and EXCHANGE directives. As a result, we anticipate that some editing of the input files will remain necessary.

2. Task 2: Data Extraction & Interface To Teamwork/SD

An initial demonstration system is currently operational which produces a CADRE Teamwork/SD structure chart from CMS source code. This demonstrates the capabilities of passes 3 and 4.

The portions of the reverse engineering software completed to date have been used to produce the necessary data for the CADRE cdif files which were then processed with CADRE's C-REV and Team*work*/SD to produce our first examples of working structure charts. A copy of a structure chart produced from an NSWC file is included with this report .

We have completed about 2/3 of the Pass 3 design, and roughly 1/2 of the implementation. Pass4 exists in its entirety. We have drafted a design document to show how we're proceeding, which I am also including in this status report.

The tasks remaining to be completed in Pass3 are the production of Module Specifications and Data Dictionary Entries in Team*work*/SD, and the generation of compool and include file hierarchies. Also remaining is the choice of software to be used for the acceptance test; the selection is to be done by mutual agreement between NSWC and GE.

3. Task 3: ENCORE/MODEL Integration Study

On August 29th a meeting was held at CRD with Noah Prywes of Computer Command and Control Company. Two possible avenues for the integration of Model and ENCORE were identified:

integration with Elementary Statement Language (ESL), or

with the Entity-Relationship database.

We requested information from 4C regarding their internal data structures, so that a decision can be made as to which avenue is preferable. We have received some of that information, but only regarding the ESL possibility. We need to see information about the Entity-Relationship graph before we can analyze the possibilities and make a decision.

GENERAL ELECTRIC COMPANY
CORPORATE RESEARCH & DEVELOPMENT
P.O. BOX 8 (BLDG KW, ROOM C247)
SCHENECTADY, NEW YORK 12301

FINANCIAL STATUS REPORT

PROJECT TITLE: CMS-2 REVERSE ENGINEERING AND ENCORE/MODEL INTEGRATION

CONTRACT NO: N00014-91-C-0240

PERIOD OF PERFORMANCE: 08/01/91 THROUGH 04/29/92

CONTRACT VALUE: $126 FUNDS AUTH: $100

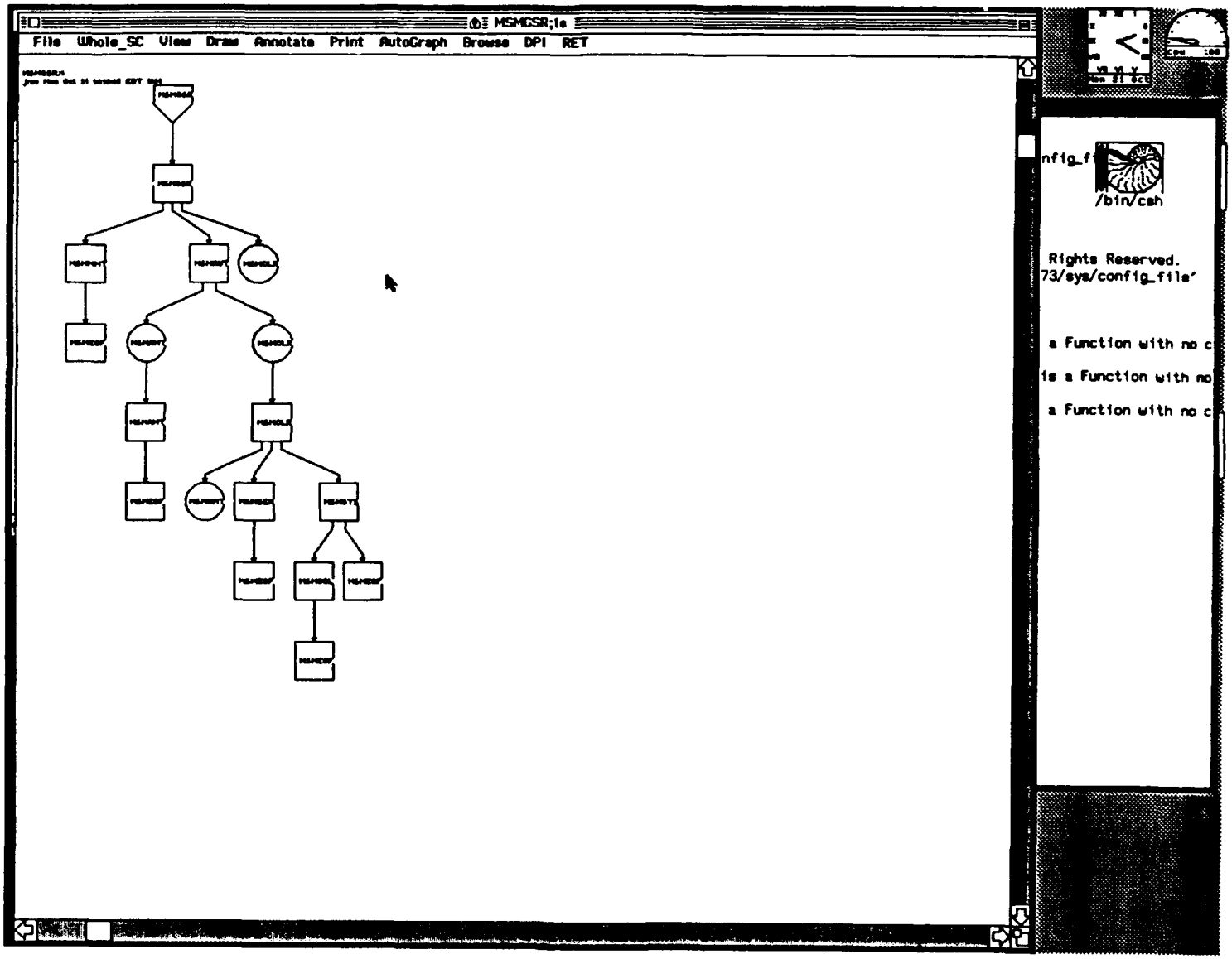
FOR PERIOD ENDING 10/29/91

C. TOTAL EXPENDITURES

Current Period	(08/01/91 - 10/27/91)	$45.
Cumulative Total to Date		$45.

COMMENTS:
Dollars in thousands

G.D. COYLE 10/31/91


MSMGSR;1s
File Whole_SC View Draw Annotate Print AutoGraph Browse DPI RET
/bin/csh
Rights Reserved.
73/sys/config_file'
a Function with no c
is a Function with no
a Function with no c

CMS2 Reverse Engineering

Pass 3 Design Document

Margaret Kelliher

1.0 This Document

This document contains the high-level and low-level designs for the pass 3 phase of the CMS reverse engineering tool. It is intended both as a working document and as a starting place for future reverse engineering tools which may want to reuse parts of our approach.

2.0 High Level Design

The purpose of pass 3 is to read in CMS-2 source, extract information from it, and write the information out to the middle files. The overall view of pass 3 consists of a main routine which handles the command line interface and drives the parser and the node processing. Node processing consists of a high-level driver which sorts out the nodes that come in, mid-level routines which further sort the nodes, and low-level routines which will do the actual processing of the nodes, ie the writing out of the middle files.

2.1 Command Line Interface

We are deriving much of our user interface from JRET's pass 3. We have not decided exactly which options we will support, but at the very least, we will include the following:

- -P filename: specifies that the following argument is a filename containing a directory search path
- -F filename : specifies that the following argument is a filename containing a list of files to process before any other input files
- -c: produces middle files for compool modules; default is just system modules
- -csci csci_name: indicates that the name specified is to be used as the default CSCI name

2.2 Parsing Strategy

We are reusing the parser from the CMS-2 translator. We will parse the COMPOOL files separately, before the main file which references them.

2.3 Processing Strategy

Our basic strategy is to divide the CMS-2 nodes into 6 basic categories, each of which will be processed in a separate package. The top-level routine will determine the category to which a node belongs, and will send it to the corresponding mid-level routine. These mid-level routines are responsible for determining whether the node is to be processed, ignored, or bypassed. If it is to be processed, a low-level routine is triggered which does the node-specific processing required.

The 6 basic categories are as follows:

- **structural statements**: generally only needed in order to access the more interesting types of statements to which they point.
- **options**: generally uninteresting, except in the way they affect our interpretation of other statements.
- **subprogram declarations**: contain the subprogram name, formal parameters, the location of its declaration, and access to its executable statements.
- **data declarations**: if global, we are interested in the declaration's location, type information, and any structures the declaration may contain. We also note any references to other global data and types.
- **executable statements**: examined for references to global data items and calls to subprograms.
- **substatement clauses** are clauses which occur only within a statement and are not themselves statements. The upper- and mid-level routines should never encounter these types of nodes, and they will be processed, as appropriate, inside the low-level routines.

For a complete breakdown of the nodes, see Appendix A.

2.4 Basic Data Structure

Our data structure is the ADL parse tree which is used in the CMS2-to-Ada translator.

3.0 Low-Level Design

3.1 Packaging Summary

The major packages will be as follows: *(italics indicate a generic package)*

(* indicates almost complete reuse; # indicates significant reuse):

- Main * (contains main driver, handling command-line interface and file control)
- Parsing *
- Lexical Analysis *

- *Parse_Control* * (helper package which deals with parser and classification routines)
- Classification (high-level node-processing routines; basically sorts them out)
- Structure_Processing (mid- and low-level routines)
- Option_Processing (mid- and low-level routines)
- Subprogram_Processing # (mid- and low-level routines)
- Data_Decl_Processing # (mid- and low-level routines)
- Executable_Processing # (mid- and low-level routines)
- *Print_Middle_File* # (language-independent printing routines; mainly utilities)
- *Source_file_database* * (associates nodes with file names and line numbers)
- *Scoping* * (keeps track of which data items are global)
- *Subprogram_Lists* # (data package for communication between Subprogram_Processing and Executable_Processing)
- System_Info (data package indicating what the current options are and what structure is being processed; set from Options_processing and Structure_Processing; get from other Processing packages; possibly to include command-line options too? or should these just be passed along as flags?)
- Symbol tables and symbol table management * (several related packages)
- CMS_records * (data structure)
- CMS_interface * (access routines for cms_records)
- *Copy_File_Handling* # (to produce copy-file hierarchy)
- *Compool_Reporting* # (to produce the compool hierarchy)

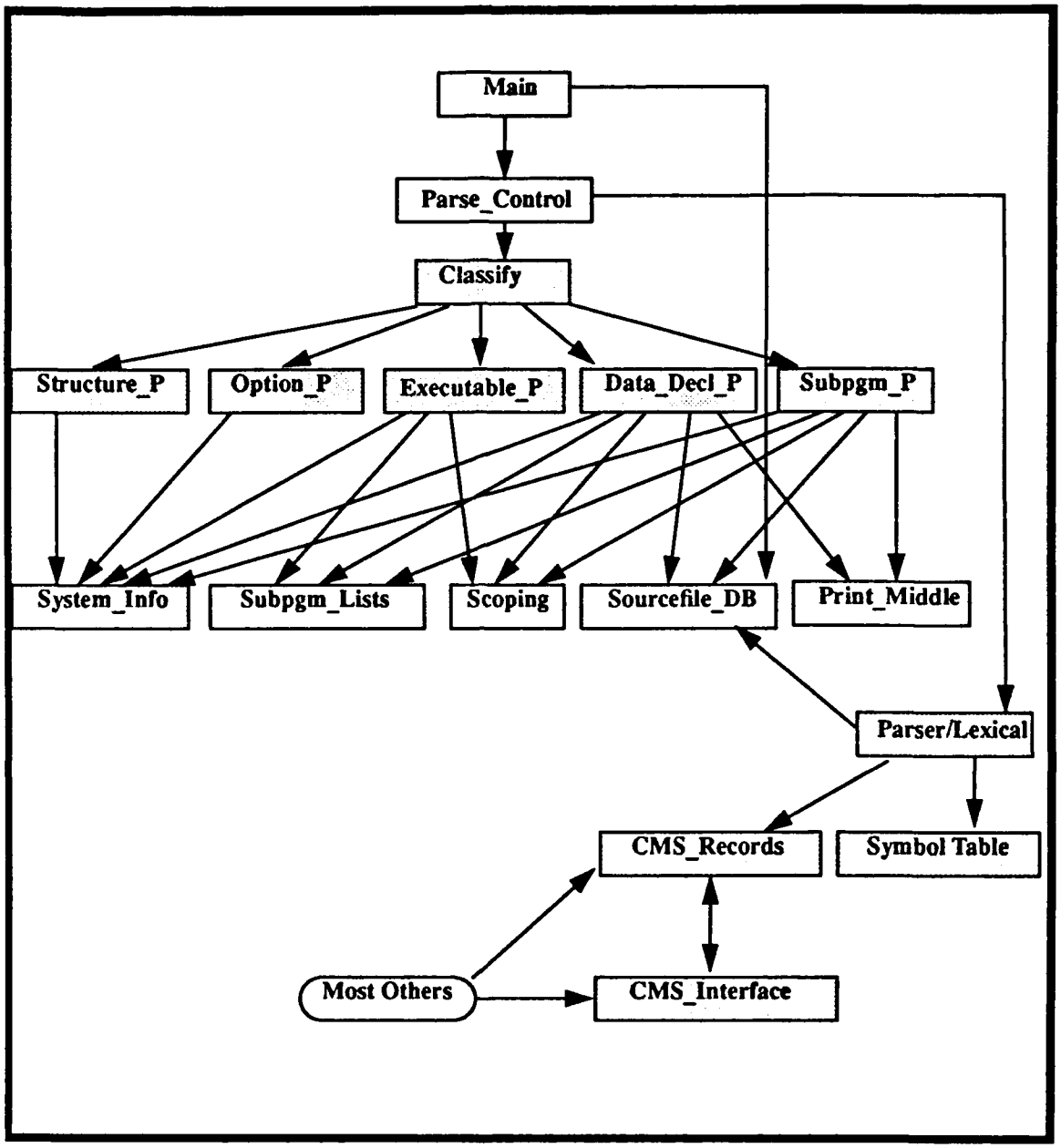


FIGURE 1. Withing Relationships between Packages

3.2 Details of Packages

3.2.1 Main Package

The Main package will contain the Main routine. The main routine will decipher the command-line parameters and control the main loop which calls the parser and the node processor once for each file specified. Most of this code can be reused from JRET.

Questions:

1. Which command-line flags do we want in the full system? All of them?
2. Which flags do we want ready for the demo (early November)?

   As few as possible as long as we can still run; probably none.
3. Which package is responsible for copy file hierarchies?

   In JRET, copy_file_handling, whose routines are called by the lexical analyzer. (Shouldn't be necessary for November)
4. Which package is responsible for compool hierarchies?

   In Jovial, extract_info. I guess we'll make some new packages to handle it. Will we have enough info (without compool nodes)? (Shouldn't be necessary by November)

   It looks like we will have compool nodes after all...
5. Will the main procedure be responsible for sending information to the Sourcefile_Database?

   In Jovial, the main procedure calls a subprogram in Sourcefile_Database, with the name of the next file to be processed. This same filename is then passed to the new initialization routine of the parser. We intend to do likewise.

   It is possible that this will be changed so that all calls come from the parser. This would make things cleaner. It is being investigated, but is low-priority...

The Procedure **Main** will be reused directly from JRET. It will call the parse_and_process routine in the parse_control package, which will call the parser and the Process_a_Node routine in the classify package with the topmost node of the parse tree.

3.2.2 Parser Package

The Parser package will contain the parser. This is being reused from the CMS-2 to Ada translator. There is still some work to be done regarding compools, cswitches and macros.

Questions:

1. The parser should interface with the Sourcefile_Database package, associating line numbers (and source files) with nodes.
2. How is the parser structured with regard to withing of packages. Is it fairly stand-alone, or are there lots of circular dependencies within the translator that need sorting out?

3. The parser is currently a function. Do we want to change it so that it is a procedure using Record_Parse_Result from the Parse_Control package?

3.2.3 Lexical Packages

The Lexical Packages will contain the lexical analysis. This is being reused from the CMS-2 to Ada translator.

Questions:

1. Do we need to add calls to copy_file_handling in the lexical analysis in order to get the copy file hierarchy?

   Yes. Not needed for November demo.

3.2.4 Parse_Control Package

The Parse_Control Package contains two subroutines: Parse_and_Extract and Record_Parse_Result. Parse_and_Extract calls the parser and then, using the resulting node, calls the classification routines. It is called by main. Record_Parse_Result is called by the parser. It stores the result of the parse into a variable which is local to the package so that it is available for Parse_and_Extract.

We will make this package generic. (the three things to be specified at instantiation are the names of the parser and parser initialization and the type of node).

(For the short term, we will also change the call to the parser so that it reflects the fact that the cms parser is a function rather than a procedure.)

3.2.5 Classification Package

The Classification package will contain the following subroutines:

The procedure **Process_A_Node** takes as input a CMS_Node (N), and probably some command-line flags. The logic is as follows:

```
if is_structural_node(N) then Process_structural_node(N, flags)
elsif is_option_node(N) then Process_option_node(N, flags)
elsif is_subprogram_decl(N) then Process_subprogram_decl(N, flags)
elsif is_data_decl(N) then Process_data_decl(N, flags)
elsif is_executable_node(N) then Process_executable_node(N, flags)
else null
endif
```

The functions **Is_Structural_Node, Is_Option_Node, Is_Subprogram_Node, Is_Data_Decl,** and **Is_Executable** all take as input a CMS_Node (N), and return true if N is of the type specified in the function name, false otherwise. They will be implemented with case statements, most likely.

The procedure **Process_Seq_of_Nodes** takes as input a sequence of CMS nodes (S) and probably some flags. The logic is as follows:

```
for each element x in S loop
Process_A_Node(x, flags)
end loop
```

The procedure **Process_Expression** takes as input a CMS_Node (N), and probably some flags. The logic is as follows:

```
case kind(N) is
    when binary_exp =>
                        Process_Expression(get_left(N), flags)
                        Process_Expression(get_right(N), flags)
    when bit_call =>    Process_Expression(get_bit_count(N), flags)
                        { ignore data_unit for now }
                        Process_Expression(get_starting_bit(N), flags)
    when built_in_func =>
                        Process_Seq_of_Expressions(
                            get_input_parameters(N), flags)
    when char_call =>   Process_Expression(get_char_count(N), flags)
                        { ignore data_unit for now }
                        Process_Expression(get_starting_char(N),flags)
    when paren_exp =>
                        Process_Expression(get_right(N), flags)
    when trailing_unary_exp =>
                        Process_Expression(get_left(N), flags)
    when unary_exp =>
                        Process_Expression(get_right(N), flags)
    when user_function_call =>
                        Process_User_Function_Call(N), flags)
    when others => null
    { references and corad/fcorad_calls will be processed in full system }
end case
```

The procedure **Process_Seq_of_Expressions** takes as input a Seq_of_CMS_Node (S), and probably some flags. The logic is as follows:

```
For all x in S loop
    Process_Expression(x, flags)
end loop
```

Questions:

1. Are we too tolerant in Process_A_Node; should we only have null for certain expected types, like Empty and Undefined, and error on unexpected ones? Similarly for Process_Expression

3.2.6 Structure_Processing Package

The Structure_Processing package will contain all the subroutines which process the nodes which are in the Structural category. They are as follows:

The procedure **Process_Structural_Node** takes as input a CMS_Node (N), and probably some flags. The logic is as follows:

```
case kind(N) is
when auto_data_design => process_auto_data_design(N, flags);
when cms_system => process_cms_system(N, flags);
(similarly for all italicized members of the structural category)
when others => null;
end case;
```

The procedure **Process_Auto_Data_Design** takes as input a CMS_Node (N), and probably some flags. The logic is as follows: Stubbed out for now (mostly data). But what about program declarations which can occur here?

The procedure **Process_CMS_System** takes as input a CMS_Node (N), and probably some flags. The logic is as follows:

```
print a group_decl for get_name(N)
Process_Node (get_major_header_block(N))
Process_Seq_of_Nodes (get_system_element_list(N))
```

The procedure **Process_Local Data_Design** takes as input a CMS_Node (N), and probably some flags. The logic is as follows: Stubbed out for now (mostly data). But what about program declarations which can occur here?

The procedure **Process_Major_Header_Block** takes as input a CMS_Node (N), and probably some flags. The logic is as follows: Stubbed out for now (only options and data).

The procedure **Process_Minor_Header_Block** takes as input a CMS_Node (N), and probably some flags. The logic is as follows: Stubbed out for now (only options and data).

The procedure **Process_Program_Body** takes as input a CMS_Node (N), and probably some flags. The logic is as follows:

```
Process_Node(get_data_definition_list(N)) {for now, a no-op}
Process_Seq_of_Nodes(get_statement_list(N)) {should be all executables}
{ Do we need to set a flag in System_Info? (Looks like no) }
```

(This could be moved over to be part of the processing of the subprogram declarations if it doesn't really warrant a procedure all its own.)

The procedure **Process_Subprogram_Data_Design** takes as input a CMS_Node (N), and probably some flags. The logic is as follows: Stubbed out for now (data only).

The procedure **Process_System_Data_Block** takes as input a CMS_Node (N), and probably some flags. The logic is as follows: Stubbed out for now (data only).

The procedure **Process_System_Data_Design** takes as input a CMS_Node (N), and probably some flags. The logic is as follows: Stubbed out for now (mostly data). But what about program declarations which can occur here?

The procedure **Process_System_Procedure_Block** takes as input a CMS_Node (N), and probably some flags. The logic is as follows:

```
Process_Node(Get_Minor_Header_Block(N), flags)
{ for now, ignore sys_proc_decl, which has name and type }
Process_Seq_of_Nodes(Get_Sys_Proc_List(Get_System_Procedure(N)),
flags)
{Do we need to set anything in Sys_Info? I don't think so}
```

Questions:

1. Are there any structural statements which will require that we perform sets in the System_Info package?

   For example, which ones trigger Start_Processing_Globals and Stop_Processing_Globals? (This can wait until November)

2. Auto-DD's , Loc-DD's and Sys_DD's can contain local and external program declarations (placeholders only). Exactly what do we do with these? (This can wait until November)

3.2.7 Option_Processing Package

The Option_Processing package will contain all the subroutines which process the nodes which are in the Options category. They are as follosws:

The procedure **Process_Option_Node** takes as input a CMS_Node(N), and probably some flags. The logic is as follows:

```
case kind(N) is
for each italicized member x of option category in Appendix A
when x => Process_X(N, Flags);
when others => null;
end case;
```

The Process_X subroutines used in Process_Option_Node will also be in this package. They will be added later.

Questions:

1. What are designs for low-level routines?
2. Which low-level routines need to be done for the November demo?

None. The whole routine will be stubbed out for the November demo.

3. Are there any option statements which will require that we perform sets in the System_-Info package?

3.2.8 Subprogram_Processing package

The Subprogram_Processing package will contain all the subroutines which process the nodes which are in the Subprogram Declaration category. They are as follows:

The procedure **Process_Subprogram_Decl** takes as input a CMS_Node(N), and probably some flags. The logic is as follows:

```
case kind(N) is
for each italicized member x of subprogram decl category in Appendix A
when x => Process_X(N, Flags);
when others => null;
end case;
```

The procedure **Process_Executive_Procedure_Block** takes as input a CMS_Node(N), and probably some flags. The logic is as follows:

```
Print_External_Indicator(N)
{How to indicate executive status? Change middle file grammar? later}
Print("procedure ", get_designator(get_decl(N))
Print_Source_Info(N)
Print_Formals(
      get_input_parameter_list(get_decl(N)),
      empty_list,
      empty_list)
Process_Node(get_program_body(N), flags)
Print_Subprogram_Lists
Print_Line("end")
```

The procedure **Process_External_Program_Declaration** takes as input a CMS_N-ode(N), and probably some flags. The logic is as follows: Stubbed out for now.

The procedure **Process_Function_Block** takes as input a CMS_Node(N), and probably some flags. The logic is as follows:

```
Print_External_Indicator(N)
Print("function ", get_designator(get_decl(N))
Print_Source_Info(N)
Print_Type(get_return_type(get_decl(N))
Print_Formals(
      get_input_parameter_list(get_decl(N)),
      empty_list,
      empty_list)
Process_Node(get_program_body(N), flags)
```

```
Print_Subprogram_Lists
Print_Line("end")
```

The procedure **Process_Local_Program_Declaration** takes as input a CMS_Node(N), and probably some flags. The logic is as follows: Stubbed out for now.

The procedure **Process_Procedure_Block** takes as input a CMS_Node(N), and probably some flags. The logic is as follows:

```
Print_External_Indicator(N)
Print("procedure ", get_designator(get_decl(N))
Print_Source_Info(N)
Print_Formals(
    get_input_parameter_list(get_formal_io_parameters(get_decl(N))),
    get_output_name_list(get_formal_io_parameters(get_decl(N)))),
    get_abnormal_exit_list(get_decl(N)))
Process_Node(get_program_body(N), flags)
Print_Subprogram_Lists
Print_Line("end")
```

The procedure **Print_External_Indicator** takes as input a CMS_Node(N). The logic is as follows:

```
if get_scope(N) = (what value?)
or else System_Info.is_processing_globals = TRUE
then Print("external ")
end if
```

The procedure **Print_Source_Info** takes as input a CMS_Node(N). The logic is as follows:

```
Print(Sourcefile_DB.get_expanded_line(N))
Print(Sourcefile_DB.get_expanded_file(N))
Print(Sourcefile_DB.get_actual_line(N))
Print(Sourcefile_DB.get_actual_file(N))
```

The procedure **Print_Formals** takes as input 3 Seq_of_CMS_Node (In, Out, and Exit). The logic is as follows:

```
if In or Out is nonempty then
    print "formals ("
    for all x in In loop
        find name and print it (with preceding comma and return if nec.)
        print (" in ")
        find type and print it (or else "unknown")
    end loop
    for all x in Out loop
        find name and print it (with preceding comma and return if nec.)
        print ("out ")
        find type and print it (or else "unknown")
```

```
        end loop
        print _line(")")
    end if
    {ignore exits for now; requires a middle file change}
```

The procedure **Print_Subprogram_Lists** takes no parameters. The logic is as follows:

```
Print_Simple_List("locals", Scope_Determination.get_locals)
        { print contexts, if we have that information}
Print_Calls
Update_Reads_and_Writes
Print_Reads
Print_Writes
Print_Reads_and_Writes
        { no nested subroutines in CMS? }
        { no header or copy files for now }
        {no pseudo code for now}
Reset_Subprogram_Lists
```

Questions:

1. What exactly is the nature of this package's interaction with the System_Info, Sourcefile_Database and Subprogram_Lists packages?

- Sourcefile_Database: get_actual_file, get_actual_line, get_expanded_file, get_expanded_line
- System_Info: if_processing_globals
- Subprogram_Lists: add_to_reads/writes/calls, get_calls/reads/writes/reads_writes/context, update_reads_and_writes, reset_subprogram_lists

2. Is the Scoping package important to this package? (directly)

   Yes, if we do scoping the way JRET does, where we keep lists of local scopes, and we have routines analogous to Enter_ and Leave_Scope. Then the scoping package can give us the list of locals. (Need push_locals, pop_locals, display_locals, is that it? Will we keep nested and locals intertwined the way JRET does? I think perhaps we should separate them out ...)

3. Can subprograms nest? If so, what do we have to do to accomodate that?

   As I read the grammar, they cannot. However, in order to be able to extend easily to other languages, we should make the Subprogram_Lists package able to handle it. See that package for more ramblings.

4. For Print_External_Indicator, what values are we looking for in get_scope?

5. For Print_Formals, how do I get the type of a parameter? From a Parameter statement, to which I hope the id_ref will point? Is there a default setting for those parameters not declared in a parameter statement?

6. In Process_Executive_Procedure, do we want to indicate executive status in the middle files? It will require a change in the grammar. (Decide after November)

7. Will we have any context statements? They should correspond to compools, but we don't have any nodes to represent them. (Joe is probably going to add them: even though they are not essential to us, they are important to the translator.) Tackle after November demo.

3.2.9 Data_Decl_Processing package

The data declaration names will be stored as name+filename+line# in order to deal with name collisions. These will be sorted out in pass4 to make the names readable. If necessary, they will be stored as name+stub+0 until the sourcefile_db is wired in.

The Data_Decl_Processing package will contain all the subroutines which process the nodes which are in the Data Declaration category. They are as follows:

The procedure **Process_Data_Decl** takes as input a CMS_Node(N), and probably some flags. The logic is as follows:

```
case kind(N) is
for each italicized member x of data decl category in Appendix A
when x => Process_X(N, Flags);
when others => null;
end case;
```

The procedure **Process_Cswitch** takes as input a CMS_Node (N), and probably some flags. The logic is as follows: Stubbed out for now.

The procedure **Process_Double_Switch_Block** takes as input a CMS_Node (N), and probably some flags. The logic is as follows: Stubbed out for now.

The procedure **Process_Equals_Declaration** takes as input a CMS_Node (N), and probably some flags. The logic is as follows:

```
{ignore designator and scoping for now}
Process_Expression(get_tag_expression(N), flags)
```

The procedure **Process_Field_Declaration** takes as input a CMS_Node (N), and probably some flags. The logic is as follows:

```
{ignore designator and scoping for now}
{ignore preset tag for now}
Process_Expression(get_repetition_count(get_initial_values(N)), flags)
Process_Expression(get_starting_bit(N), flags)
{ignore type for now}
Process_Expression(get_word_number(N), flags)
```

The procedure **Process_Field_Overlay_Declaration** takes as input a CMS_Node (N), and probably some flags. The logic is as follows:

```
{ignore field_name for now}
For all x in get_sibling_list(N) loop
```

```
        if kind(x) = expression
            then Process_Expression(x, flags)
            { else ignore for now }
        end if
    end loop
```

The procedure **Process_Format_Declaration** takes as input a CMS_Node(N), and probably some flags. The logic is as follows:

```
{ignore designator, scoping and modifiers for now}
for each x in get_format_list(N) loop
    Process_Format_Item(x, flags)
end loop;
```

The procedure **Process_Format_Item** takes as input a CMS_Node(N), and probably some flags. The logic is as follows:

```
case kind(N) is
    when repeated_format =>
                    Process_Expression(get_count(N), flags)
                    Process_Format_Item(
                        get_field_descriptor(N), flags)
                    for all x in get_format_list(N) loop
                        Process_Format_Item(x, flags)
                    end loop
    when format_descriptor =>
                    Process_Expression(get_field_width(N), flags)
                    Process_Expression(get_fraction_size(N), flags)
    when double_format_item =>
                    Process_Format_Item(
                        get_current_format_item(N), flags)
                    Process_Format_Item(
                        get_next_format_item(N), flags)
    when format_positioner => {both x and t}
                    Process_Expression(get_count(N), flags)
    when others =>  null
end case
```

The procedure **Process_Index_Switch_Block** takes as input a CMS_Node (N), and probably some flags. The logic is as follows: Stubbed out for now.

The procedure **Process_Item_Area_Declaration** takes as input a CMS_Node (N), and probably some flags. The logic is as follows: Stubbed out for now.

The procedure **Process_Item_Switch_Block** takes as input a CMS_Node (N), and probably some flags. The logic is as follows: Stubbed out for now.

The procedure **Process_Loadvrbl_Declaration** as input a CMS_Node(N), and probably some flags. The logic is as follows:

```
{ignore designator, scoping and type for now}
Process_Expression(get_initial_value(N), flags)
```

The procedure **Process_Local_Index** takes as input a CMS_Node (N), and probably some flags. The logic is as follows: Stubbed out for now.

The procedure **Process_Nitems_Declaration** takes as input a CMS_Node(N), and probably some flags. The logic is as follows:

```
{ignore designator, scoping and type for now}
Process_Expression(get_initial_value(N), flags)
```

The procedure **Process_Overlay_Declaration** takes as input a CMS_Node (N), and probably some flags. The logic is as follows:

```
{ignore data_unit for now}
For all x in get_sibling_list(N) loop
    if kind(x) = expression
        then Process_Expression(x, flags)
        { else ignore for now }
    end if
end loop
```

The procedure **Process_Parameter_Declaration** takes as input a CMS_Node(N), and probably some flags. The logic is as follows:

```
{ignore designator, modifier, scoping and type for now}
case kind(get_initial_value(N)) is
    when expression => Process_Expression(get_initial_value(N), flags)
    when preset_tag_with_magnitude =>
                        Process_Expression(
                            get_preset_value(get_initial_value(N)), flags)
                        Process_Expression(
                            get_magnitude(get_initial_value(N)), flags)
                        Process_Expression(
                            get_bit_position(get_initial_value(N)), flags)
    when others => null
end case
```

The procedure **Process_Pdouble_Switch_Block** takes as input a CMS_Node (N), and probably some flags. The logic is as follows:

```
{ignore designators and modifiers for now}
Add_to_Calls(N)
```

The procedure **Process_Pindex_Switch_Block** takes as input a CMS_Node (N), and probably some flags. The logic is as follows: stubbed out for now.

The procedure **Process_Pitem_Switch_Block** takes as input a CMS_Node (N), and probably some flags. The logic is as follows: stubbed out for now.

The procedure **Process_Scaled_Data_Unit** takes as input a CMS_Node(N), and probably some flags. The logic is as follows: stubbed out for now.

The procedure **Process_Simple_Type_Decl** takes as input a CMS_Node(N), and probably some flags. The logic is as follows: stubbed out for now.

The procedure **Process_Structured_Type_Decl** takes as input a CMS_Node(N), and probably some flags. The logic is as follows:

```
{ignore designator and visibility for now}
Process_Expression(get_packing(N))
For all x in get_structure_information_list(N) loop
    case kind(x) is
        when field_declaration(x) =>
                Process_Node(x, flags)
        when field_overlay_declaration =>
                Process_Node(x, flags)
        when range_declaration => {ignore name for now}
                Process_Expression(get_upper_limit(x), flags)
                Process_Expression(get_lower_limit(x), flags)
    end case
end loop
```

The procedure **Process_Sub_Table_Declaration** takes as input a CMS_Node(N), and probably some flags. The logic is as follows:

```
{ignore designator, major index, modifier, and parent table for now}
if kind(get_number_of_items(N)) = expression
    then Process_Expression(get_number_of_items(N), flags)
    { else ignore for now }
end if
Process_Expression(get_starting_item_number(N), flags)
```

The procedure **Process_System_Index_Declaration** takes as input a CMS_Node(N), and probably some flags. The logic is as follows:

```
for all x in get_system_index_list(N) loop
    {ignore designator for now}
    Process_Expression(get_register_number(x), flags)
end loop
```

The procedure **Process_Table_Block** takes as input a CMS_Node(N), and probably some flags. The logic is as follows:

```
{ ignore designator, indirect indicator, major index, modifier,
    and table form for now}
Process_Expression(get_dimension_list(N), flags)
for all x in get_table_list(N) loop
    case kind(x) is
        when field_declaration(x) =>
```

```
                    Process_Node(x, flags)
            when field_overlay_declaration(x) =>
                    Process_Node(x, flags)
            when range_declaration => {ignore name for now}
                    Process_Expression(get_upper_limit(x), flags)
                    Process_Expression(get_lower_limit(x), flags)
            when like_table_declaration => {ignore all else for now}
                    Process_Expression
                        (get_number_of_items(x), flags)
            when item_area_declaration => {ignore for now}
            when sub_table_declaration =>
                    Process_Node (x, flags)
        end case
    end loop
```

The procedure **Process_Variable_Declaration** takes as input a CMS_Node(N), and probably some flags. The logic is as follows:

```
{ignore designator, modifier, scoping and type for now}
case kind(get_initial_value(N)) is
    when expression => Process_Expression(get_initial_value(N), flags)
    when preset_tag_with_magnitude =>
                    Process_Expression(
                        get_preset_value(get_initial_value(N)), flags)
                    Process_Expression(
                        get_magnitude(get_initial_value(N)), flags)
                    Process_Expression(
                        get_bit_position(get_initial_value(N)), flags)
    when others => null
end case
```

Questions:

1. What exactly is the nature of this package's interaction with the System_Info, Sourcefile_Database and Scoping packages?

   System_Info: should be query-only.(if_processing_globals)

   Sourcefile_Database: get_expanded_file_name, get_expanded_line_number, get_actual_file_name, get_actual_line_number.

   Scoping: tell about each declaration we come across so that it can enter it into the appropriate scope, except for the outermost level. (add_to_locals, add_to_params, is that it?)

2. Will we need Subprogram_List.Add_to_Reads for type declarations?

   (Can be decided after November demo)

3. Should procedure switch blocks count as calls to procedures?

   Currently, I am considering the call to occur in procedure call phrase instead.

3.2.10 Executable_Processing package

The Executable_Processing package will contain all the subroutines which process the nodes which are in the Executable Statements category. They are as follosws:

The procedure **Process_Executable_Statment** takes as input a CMS_Node(N), and probably some flags. The logic is as follows:

```
case kind(N) is
for each italicized member x of executable stmt category in Appendix A
when x => Process_X(N, Flags);
when others => null;
end case;
```

The procedure **Process_Begin_Block** takes as input a CMS_Node (N), and probably some flags. The logic is as follows:

```
{ignore the labels}
Process_Seq_of_Nodes(get_statement_list(N), flags)
```

The procedure **Process_Cswitch_Off** takes as input a CMS_Node (N), and probably some flags. The logic is as follows: Stubbed out for now.

The procedure **Process_Cswitch_On** takes as input a CMS_Node (N), and probably some flags. The logic is as follows: Stubbed out for now.

The procedure **Process_Data_Statement** takes as input a CMS_Node (N), and probably some flags. The logic is as follows: Stubbed out for now.

The procedure **Process_Debug_Decl** takes as input a CMS_Node (N), and probably some flags. The logic is as follows: Stubbed out for now.

The procedure **Process_Display_Phrase** takes as input a CMS_Node (N), and probably some flags. The logic is as follows: Stubbed out for now.

The procedure **Process_End_Trace_Phrase** takes as input a CMS_Node (N), and probably some flags. The logic is as follows: Stubbed out for now.

The procedure **Process_Exec_Phrase** takes as input a CMS_Node (N), and probably some flags. The logic is as follows:

```
{ ignore labels }
Process_Expression(Get_parameter_1(N), flags)
Process_Expression(Get_parameter_2(N), flags)
```

The procedure **Process_Exit_Phrase** takes as input a CMS_Node (N), and probably some flags. The logic is as follows: Stubbed out for now.

The procedure **Process_Find_Statement** takes as input a CMS_Node (N), and probably some flags.The logic is as follows:

```
{ignore the if_data_clause}
Process_Node(get_else_clause(N),flags)
Process_Expression(get_find_condition(N), flags)
Process_Node(get_imperative_statement(N), flags)
Process_Expression(
      get_increment(get_by_clause(get_control_clause(
            get_varying_clause(N))))), flags)
Process_Expression(
      get_limit(get_thru(get_control_clause(get_varying_clause(N)))),
      flags)
{ignore within part of control_clause for now}
Process_Expression(
      get_initial_value(get_from_clause(get_control_clause(
            get_varying_clause(N))), flags)
{ignore data_unit of varying_clause for now}
```

The procedure **Process_For_Block** takes as input a CMS_Node (N), and probably some flags. The logic is as follows:

```
Process_Expression(get_expression(N), flags)
Process_Node(get_else_clause(N), flags)
{ignore labels and types for now}
for all x in get_value_block_list(N) loop
      {ignore labels}
      Process_Seq_of_Expressions (get_value_list(x), flags)
      Process_Seq_of_Nodes(get_statement_list(x))
end loop
```

The procedure **Process_Function_Return** takes as input a CMS_Node (N), and probably some flags. The logic is as follows:

```
Process_Expression(get_value(N), flags)
```

The procedure **Process_If_Statement** takes as input a CMS_Node (N), and probably some flags. The logic is as follows:

```
Process_Expression(get_conditional_expression(N), flags)
Process_Node(get_else_clause(N), flags)
for all x in get_elsif_clause_list(N) loop
      Process_Expression(get_conditional_expression(x), flags)
      Process_Node(get_imperative_statement(x), flags)
end loop
Process_Node(get_imperative_statement(N), flags)
```

The procedure **Process_Imperative_Statement** takes as input a CMS_Node (N), and probably some flags. The logic is as follows:

```
Process_Seq_of_Nodes(get_simple_statement_list(N), flags)
```

The procedure **Process_Index_Goto_Phrase** takes as input a CMS_Node (N), and probably some flags. The logic is as follows:

```
{ ignore invalid and labels }
Process_Expression(get_selector(N), flags)
{ignore special condition}
{stub out switch name for now}
```

The procedure **Process_Item_Goto_Phrase** takes as input a CMS_Node (N), and probably some flags. The logic is as follows: Stubbed out for now.

The procedure **Process_Pack_Phrase** takes as input a CMS_Node (N), and probably some flags. The logic is as follows: Stubbed out for now.

The procedure **Process_Pindex_Call_Phrase** takes as input a CMS_Node (N), and probably some flags. The logic is as follows:

```
Process_Expression(get_control(N), flags)
if get_switch_name(N) is a pdouble_switch_block then
      for all x in (get_pdouble_list(get_switch_name(N))) loop
            Subprogram_Lists.Add_to_Calls(get_first_proc(x))
            Subprogram_Lists.Add_to_Calls(get_second_proc(x))
      end loop
else {must be a pindex_switch_block}
      for all x in (get_pindex_list(get_switch_name(N))) loop
            Subprogram_Lists.Add_to_Calls(x)
      end loop
end if
{ignore invalid, labels and parameters for now}
```

The procedure **Process_Pitem_Call_Phrase** takes as input a CMS_Node (N), and probably some flags. The logic is as follows:

```
{ ignore invalids, labels }
Process_Seq_of_Expressions(get_input(get_parameters(N)), flags)
for all x in (get_pitem_list(get_switch_name(N)) loop
      Add_to_Calls(get_proc_name(x))
end loop
```

The procedure **Process_Procedure_Return** takes as input a CMS_Node (N), and probably some flags. The logic is as follows: Stubbed out for now.

The procedure **Process_Set_Phrase** takes as input a CMS_Node (N), and probably some flags. The logic is as follows:

```
Process_Expression(get_source(N), flags)
{ignore labels, overflow, remainder, and targets for now}
```

The procedure **Process_Shift_Phrase** takes as input a CMS_Node (N), and probably some flags. The logic is as follows:

Process_Expression(get_shift_count(N), flags)
{ignore data_unit, direction, labels, shift_type, and target for now}

The procedure **Process_Simple_Goto_Phrase** takes as input a CMS_Node (N), and probably some flags. The logic is as follows: Stubbed out for now.

The procedure **Process_Snap_Phrase** takes as input a CMS_Node (N), and probably some flags. The logic is as follows:

Process_Expression(get_magnitude(get_preset_magnitude(N)), flags)
Process_Expression(get_bit_position(get_preset_magnitude(N)), flags)
{ignore data_unit and labels for now}

The procedure **Process_Substitution_Decl** takes as input a CMS_Node (N), and probably some flags. The logic is as follows: Stubbed out for now.

The procedure **Process_Supplied_Procedure_Call_Phrase** takes as input a CMS_Node (N), and probably some flags. The logic is as follows:

{ ignore labels and procedure name }
Process_Seq_Of_Expressions(get_input(get_parameters(N)), flags)
{ ignore ouput parameters for now }

The procedure **Process_Swap_Phrase** takes as input a CMS_Node (N), and probably some flags. The logic is as follows: Stubbed out for now.

The procedure **Process_Trace_Phrase** takes as input a CMS_Node (N), and probably some flags. The logic is as follows: Stubbed out for now.

The procedure **Process_User_Function_Call** takes as input a CMS_Node (N), and probably some flags. The logic is as follows:

Subprogram_Lists.Add_To_Calls(get_function_name(N))
Process_Seq_Of_Expressions(get_input_parameters(N), flags)

The procedure **Process_User_Procedure_Call_Phrase** takes as input a CMS_Node (N), and probably some flags. The logic is as follows:

Subprogram_Lists.add_to_calls(get_proc_name(N))
Process_Seq_Of_Expressions(get_input(get_parameters(N)), flags)
{ ignore labels, exit parameters, and ouput parameters for now }

The procedure **Process_Expression** takes as input a CMS_Node (N), and probably some flags. The logic is as follows:

```
case kind(N) is
    when binary_exp =>
                        Process_Expression(get_left(N), flags)
                        Process_Expression(get_right(N), flags)
    when bit_call =>    Process_Expression(get_bit_count(N), flags)
                        { ignore data_unit for now }
```

```
                    Process_Expression(get_starting_bit(N), flags)
        when built_in_func =>
                    Process_Seq_of_Expressions(
                        get_input_parameters(N), flags)
        when char_call =>  Process_Expression(get_char_count(N), flags)
                    { ignore data_unit for now }
                    Process_Expression(get_starting_char(N),flags)
        when paren_exp =>
                    Process_Expression(get_right(N), flags)
        when trailing_unary_exp =>
                     Process_Expression(get_left(N), flags)
        when unary_exp =>
                    Process_Expression(get_right(N), flags)
        when user_function_call =>
                    Process_User_Function_Call(N), flags)
        when others => null
        { references and corad/fcorad_calls will be processed in full system }
end case
```

Questions:

1. What exactly is the nature of this package's interaction with the System_Info, Scoping and Subprogram_Lists packages?

   System_Info: Query-only

   Scoping: is_global

   Subprogram_Lists: add_to_reads; add_to_writes; add_to_calls

3.2.11 Print_Middle_File package

The Print_Middle_File package will contain utility routines to assist the Process_* packages in their printing. We had considered a template approach in which all the printing was controlled in this package, using data from the other packages, but that quickly got very cumbersome, and didn't seem to have much benefit in this case. Instead we will have "helper" routines which know the format for certain types of items (expanded vs unexpanded variable names, for instance) and different types of lists. We will feel free to add new ones as needed. We intend this to be a generic package, but will adjust that intention as required.

The procedure **Print_Simple_List** takes as input a string (Label) and a Seq_of_Symbols (S). The logic is as follows:

```
if S is non-empty then
    Print(Label, "(")
    for each x in S loop
        Print (x) {with preceding comma, if necessary}
```

```
        end loop
        Print_line(")")
    end if
```

Questions:

1. What procedures are needed?

- Print_Expanded_Identifier (esp. for variables)
- Print_Unexpanded_Identifier (esp. for subroutines)
- Print_List_of_Expanded_Identifiers
- Print_List_of_Unexpanded_Identifiers

3.2.12 Sourcefile_Database package

The Sourcefile_Database package will keep track of which source files are currently being processed, and will calculate expanded files and line numbers for us. We will reuse as much as possible from JRET.

Questions:

1. What subroutines are in this package?

- set_top_level_source_file_name
- get_top_level_source_file_name
- get_top_level_source_file_symbol
- get_top_level_file_basename
- hold_source_info (parser will use)
- attach_source_info (parser will use)
- get_expanded_file_name (Data_Decl_Processing and Subprogram_Processing)
- get_expanded_line_number (Data_Decl_Processing and Subprogram_Processing)
- get_actual_file_name (Data_Decl_Processing and Subprogram_Processing)
- get_actual_line_number (Data_Decl_Processing and Subprogram_Processing)

2. Are there any alterations we need to make to this package?

   Want to make it generic. Passing in Node type, EQ function, and possibly Default node value. Also, want to change it so it uses the generic hashing routines.

3.2.13 Scoping package

The Scope_Determination package will keep track of which data items are global and which are local. We will reuse as much as possible from JRET (Most comes from extract_info). I had originally intended the Scope_Determination package to contain all

the language-dependent scoping rules, but JRET seems to allow the data-declaration routines to take care of that, and Scope_Determination just keeps track of those decisions. This is reasonable, and will allow Scope_Determination to be more language-independent (in fact it should be fully language-independent, except for the type of nodes, that's why we'll make it generic).

We still probably want to have a separate procedure (here or elsewhere), which, given a declaration (and maybe the System_Info package), determines whether it is local or global. (The idea is to keep these rules as contained as possible.)

Questions:

1. What subroutines are in this package?

   Probably want the following:

- push_locals, pop_locals(renamed?) (Subprogram_Processing)
- locals_display (Subprogram_Processing)
- add_to_locals (Data_Decl_Processing)
- add_to_params (Data_Decl_Processing)
- is_global (Executable_Processing)
- is_local (unused for now?)
- HASH (will be passed in as a function to the generic)
- params_node, params_seq, locals_node, locals_seq (Data_Decl_Processing?)

2. What structural statements' data declarations are by definition global (SYS-DD, COMPOOL?, ...)
3. Do we need to change this package?

   Want to make it generic

4. What other packages will this package need to communicate with?

   Subprogram_Processing, Data_Decl_Processing, Executable_Processing

5. Is there a better name for this package?

3.2.14 Subprogram_Lists package

The Subprogram_Lists package will serve to aid communication between the Subprogram_Processing and Executable_Processing packages. It will maintain the read, write, read/write, formals and calls lists. We will reuse as much as possible from JRET. We will use the Seq package (part of ADL) to build our lists.

The **Calls_List, Reads_List, Writes_List,Reads_Writes_List** are each a linked list of declaration nodes. We shall provide the following visible routines:

> The procedure **Add_to_Calls** takes as input a CMS_node (N). It gets the name (a symbol) and adds it to **Calls_List**.

The procedure **Add_to_Reads** takes as input a CMS_node (N). It gets the name (a symbol) and adds it to **Reads_List.**

The procedure **Add_to_Writes** takes as input a CMS_node (N). It gets the name (a symbol) and adds it to **Writes_List.**

The function **Print_Calls** takes no input and returns no output. It prints out the **Calls_List,** if one exists. (It must handle switch blocks as well as "regular" procedure and function calls.)

The function **Print_Reads** takes no input and returns no output. It prints out the **Reads_List,** if one exists.

The function **Print_Reads_and_Writes** takes no input and returns no output. It prints out the **Reads_Writes_List,** if one exists.

The function **Print_Writes** takes no input and returns no output. It prints out the **Writes_List,** if one exists.

The procedure **Update_Reads_and_Writes** takes no input and returns nothing. It looks for symbols which appear on both the **Reads_List** and the **Writes_List,** and removes them to the **Reads_Writes_List.**

Questions:

1. Which lists do we need to keep track of?

   Locals will be taken care of by the Scope_Determination package.

   Formals will be taken care of in Subprogram_Processing

   Reads, writes, reads_writes, and calls will be taken care of here

   What about Macros, and Contexts? Ignored for now.

2. How do we handle nested subprograms? Should they be tightly bound to locals (as in JRET) or done separately?

   I think CMS2 doesn't have nested subprograms. However, I think we need to handle this issue anyway (at least from a design standpoint; we could implement only the simple case, leaving an easy way to extend it.)

   I think that lists that we keep could be kept in hash tables, where the key would be the subprogram_decl and the value would be the read list (depending on which hash table was queried). The nested subprogram list(s) could be kept in this package along with the others that we're already keeping.

   Alternatively, we could just use stacks, pushing and popping as we start and end the subprograms.

3. What other packages will this package need to communicate with?

- Executable_Processing: add_to_reads, add_to_writes, add_to_calls

- Subprogram_Processing: update_reads_and_writes, reset_subprogram_lists, print_-reads, print_writes, print_reads_and_writes, print_calls
- Print_Middle_File: we'll want to use those utilities.

4. See Declaration_Processing for discussion of how we will store names...

   May not be pertinent to this package, as the printing utilities should hide it...

3.2.15 System_Info package

The System_Info package will serve to aid communication between the Options_ and Structure Processing packages on the one hand and the Subprogram_, Data_ and Executable_Processing packages on the other. It will maintain information about the state of the system which will affect the interpretation of other nodes.

Questions:

1. Exactly which information needs to be maintained here?

- Globals processing - ie are we in a segment of code where all declarations are by default external? (Start_Processing_Globals, Stop_Processing_Globals, If_Processing Globals)
- More as needed

2. What packages will this package need to communicate with?

- Data_Processing, Executable_Processing, Subprogram_Processing: If_processing_globals
- Structure_Processing: Start_Processing_Globals, Stop_Processing_Globals

3. If the only thing to be in here is is_processing_globals, perhaps that should go into Scope_Determination, and this package should be eliminated?

   To be decided after November demo.

3.2.16 CMS_Records package

The CMS_records package defines the ADL structure which will serve as the underpinning of our system. We will reuse it from the CMS-2 to Ada translator.

Questions:

1. Are there any changes needed?

3.2.17 CMS_Interface package

The CMS_interface package allows us to access the ADL structure which serves as the underpinning of our system. We will reuse it from the CMS-2 to Ada translator.

Questions:

1. Are any changes necessary?

   Yes. New_CMS_Node needs to initialize the fields to the Empty_Node where they are currently being left as the null pointer.

3.2.18 Symbol Table packages

We will reuse the CMS-2 to Ada translator's symbol tables and searches.

Questions:

1. Are any changes necessary?

Appendix A: Node Categories

Structural

auto_data_design
cms_system
direct_code_block
local_data_design
major_header_block
minor_header_block
program_body
subprogram_data_design
system_data_block
system_data_design
system_procedure_block

Options

cmode_decl
address_counter_separation_decl
allocataion_information
cmp_object_spec
cnv_object_spec
crg_object_spec
crl_object_spec
cr_object_spec
sadump_object_spec
sa_object_spec
scrg_object_spec
scrl_object_spec
scr_object_spec
sm_object_spec
coll_option
cswitch_delete_decl
executive_declaration
farmode_option
hex_option
independent_option
level_option
line_option
mode_field_declaration
mode_vrbl_declaration
monitor_option
mscale_option
nonrt_option
object_option
optimize_option
options_declaration
parameter_passing_decl
pooling_declaration
scale_mode_declaration
source_option
single_precision_decl
structured_option

(To be reveiwed)

Subprogram Declarations

exec_proc_block
external_program_decl
function_block
local_program_decl
procedure_block

Data Declarations

cswitch
double_switch_block
equals_declaration
field_declaration
field_overlay_decl
format_declaration
index_switch_block
inputlist_decl
io_phrase
item_area_decl
item_switch_block
loadvrbl_decl
local_index
nitems_declaration
nonstandard_file_decl
outputlist_decl
overlay_declaration
parameter_decl
pdouble_switch_block
pindex_switch_block
pitem_switch_block
scaled_data_unit
simple_type_decl
stringform_decl
structured_type_decl
sub_table_decl
standard_file_decl
system_index_decl
table_block
variable_decl

Executable Statements

begin_block
cswitch_off
cswitch_on
data_statement
debug_decl
display_phrase
end_trace_phrase
exec_phrase
exit_phrase
find_statement
for_block
function_return
if_statement
imperative_statement
index_goto_phrase
item_goto_phrase
null_phrase
pack_phrase
pindex_call_phrase
pitem_call_phrase
resume_phrase
procedure_return
set_phrase
shift_phrase
simple_goto_phrase
snap_phrase
stop_phrase
substitution_decl
supplied_procedure-_call_phrase
swap_phrase
trace_phrase
user_function_call
user_procedure_call-_phrase

(For Nov, s/r's only)

Substatement Clauses

elsif_clause
actual_parameters
binary_exp
bit_call
boolean_type
build_in_func
by_clause
character_type
char_call
corad_call
component_ref
control_clause
direct_ref
double_switch_item
double_format_item
end_switch
end_cswitchs
exec_proc_decl
fcorad_call
file_specification
fixed_type
float_type
formal_io_parameters
format_descriptor
format_list
from_clause
function_decl
indexed_ref
index_clause
integer_type
item_switch_item
like_table_option
local_data_decl
major_header
minor_header
named_ref
paren_exp
pdouble
pitem
preset_item
preset_magnitude
preset_with_magnitude
procedure_declaration
range_declaration
repeated_format

Substatement Clauses (ctd)

repeated_stringform

spill_declaration

star_data_unit

status

status_type

stringform_descriptor

stringform_list

stringform_positioner

subprogram_data_definition_list

sys_proc_declaration

system_index

system_procedure

thru_clause

trailing_unary_exp

t_format_positioner

two_word_initializer

unary_exp

unit_with_magnitude

value_block

vary_block

within_clause

x_format_descriptor

(For November, only s/r's checked)

In the above table, *italics* indicate items which are to be processed in the full system, and ***bold italics*** indicates items to be processed in the November demo system.